SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 33–46 (1997)

Research

On Program Dicing

T. Y. CHEN*,** AND Y. Y. CHEUNG
Department of Computer Science, University of Melbourne, Parkville 3052, Australia

SUMMARY

Since it is sometimes difficult to apply the technique of static program dicing in debugging programs, we introduce dynamic dicing as the dynamic counterpart of static dicing. As the effectiveness of program dicing techniques depends on the size of the program dices used, this paper uses a probabilistic approach to investigate the relationship between the size of static and dynamic program dices. In making a comparison between dynamic and static program dicing, we present and prove six propositions. This leads us to the conditions under which a dynamic program dice is smaller than a static program dice. Based on those findings, we offer five strategies for constructing dynamic program dices. © 1997 by John Wiley & Sons, Ltd. J. Software Maintenance 9: 33–46, 1997

(No. of Figures: 5. No. of Tables: 0. No. of Refs: 13.)

KEY WORDS: debugging; dynamic dicing; dynamic analysis; program dicing; program partitioning techniques; program slicing

1. INTRODUCTION

Among the various approaches to program debugging, the most common ones include core dumps at the time of failure, induction, deduction, program instrumentation, breakpoints and backtrackings. In this paper, we investigate an approach that attempts to identify the statements that are likely to contain faults. Obviously, the productivity of debugging can be significantly improved if only a subset, rather than the whole program, is to be debugged. A typical example of such an approach is the technique of program slicing. In this paper, we will investigate another technique known as program dicing which has evolved from the technique of program slicing.

The notion of static program dicing, introduced by Lyle and Weiser (1987), is based on the notion of static program slicing (Weiser, 1984) that has been used in program maintenance (Gallagher and Lyle, 1991), module cohesion analysis (Ott and Thuss, 1989), function recovery (Lanubile and Visaggio, 1993), and interprocedural data flow testing (Kamkar, Fritzson and Shahmehri, 1993). For a variable at a specified location in a program, its static program slice is defined as the set of program statements that may

* Correspondence to: T. Y. Chen, Department of Computer Science, University of Melbourne, Parkville 3052, Australia
** Current address: Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong

CCC 1040–550X/97/010033–14$17.50

Received 30 December 1995
Revised 2 October 1996

affect its value at that location. Therefore, with the exception of a missing statement fault, the static program slice of the incorrectly computed variable, must contain the fault.

With respect to a given test suite, a variable is defined as a correct variable if it is correctly computed for all elements of this test suite; otherwise it is defined as an incorrect variable. A static program dice is defined as the set difference between the static slice of an incorrect variable and that of a correct variable. Obviously, a static program dice is included in the static program slice of the corresponding incorrect variable. Hence, static program dicing is considered to be better than static program slicing as a debugging technique, as it reduces the size of the set of statements that potentially contains the fault.

Since static dicing can only be used when there is only one output variable or all output variables are incorrect, Chen and Cheung (1993) introduced dynamic program dicing to remedy this situation. As a dynamic counterpart to static program slicing, dynamic program dicing is based on dynamic slicing rather than static slicing. Since a dynamic slice only contains those statements that may affect the value of a variable at a specified location in an execution path, obviously it is included within the corresponding static slice (Gopal, 1991; Agrawal and Horgan, 1990; Korel and Laski, 1988, 1990). However, a dynamic dice is not necessarily included within the corresponding static dice. Hence, a dynamic dice may be larger in size than a static dice. Therefore, we are interested to know the conditions under which a dynamic program dice is larger in size than the corresponding static program dice, or vice versa.

This paper is organized as follows. The background of program slicing and static program dicing is given in next section. Then, the motivation and the methodology of dynamic program dicing are discussed in the third section. In the fourth section, a comparison of dynamic and static program dicing is given. Some strategies for constructing dynamic program dicing are proposed and discussed in the fifth section. The sixth section concludes the paper.

2. BACKGROUND

2.1. Program slicing

Program slicing was first introduced by Weiser (1984), who defined it as follows:

> 'Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a 'slice,' is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior.'

Although the above definition requires program slices to be executable, there are other definitions (Gopal, 1991; Agrawal and Horgan, 1990) that do not impose such a requirement on program slices. The reason is simply that in the context of debugging, we are more interested in the existence of faults in program slices than in whether the program slices are executable or not.

According to Weiser (1984), a slicing criterion C, of a program $\mathcal{P}$, is a tuple (s,V) where s is a statement in $\mathcal{P}$ and V is a subset of the variables in $\mathcal{P}$. The program slice satisfying criterion (s,V) is a subset of program statements that influences the values of

variables in V at s. The concept of program slicing introduced by Weiser (1984) is referred to as static program slicing, which captures the computation for any input rather than a specific input.

In contrast to static program slicing, dynamic program slicing is based on a program trajectory rather than the program itself. A subpath that has actually been executed for certain input is known as a program trajectory. Note that a trajectory can be a subpath of either a terminated or non-terminated execution path. The trajectory with the input ($n = 2$, $x[1] = -1$, $x[2] = 1$) for program 'Sample1' in Figure 1 is $\langle s1,s2,s3,s4,s6,s7,s2,s3,s5,s6,s7,s2,s8\rangle$. Since a statement may appear more than once in a trajectory, each element of the trajectory is represented by i^j, where statement i is at position j in the trajectory. Thus, the above trajectory is more formally expressed as $\langle s1^1,s2^2,s3^3,s4^4,s6^5,s7^6,s2^7,s3^8,s5^9,s6^{10},s7^{11},s2^{12},s8^{13}\rangle$.

A dynamic slice is a part of the program thats behaviour is the same as that of the original program with respect to a set of variables V at position s^q in the trajectory with input X. Thus, the criterion for dynamic slicing is denoted as $C = (X,s^q,V)$. Hence, the aim behind dynamic slicing is to find statements in the trajectory rather than to find a program satisfying the slicing criteria.

In this paper, we regard both static and dynamic program slices to be sets of program statements rather than executable programs. Furthermore, for the sake of clarity, we restrict our discussion to the singleton variable set V. It should be obvious that our results are also true when V contains more than one element.

2.2. Static program dicing

Lyle and Weiser (1987) introduced static program dicing as a debugging technique. This methodology is aimed at the identification of the statements in a slice that are more likely to contain a fault. We define correct and incorrect variables as variables that have their output values computed correctly and incorrectly with respect to the given test suite. A test suite is said to be reliable if it can reveal a fault whenever it exists in the program (Howden, 1976). In practice, it is extremely rare, if not impossible, to have a reliable test suite. When the test suite is not reliable, correctly computed variables may in fact be incorrect, and hence correct variables should be regarded as apparently correct variables.

```
        Program Sample1 (Input x[MAX], n : integer)
        Begin
s1          i = 1
s2          While (i ≤ n) do
s3              If (x[i] < 0)
s4                  y = f1(x[i])
                Else
s5                  y = f2(x[i])
                Endif
s6              z = f3(y)
s7              i = i+1
            Endwhile
s8          print(y, z)
        End
```

Figure 1. Program 'Sample1'

As defined by Lyle and Weiser (1987) the construction of a static program dice involves removing those statements of a static program slice for an apparently correct variable, from the static program slice for an incorrect variable. Hence, a static dice is effectively the set difference between the static slices of an incorrect variable and an apparently correct variable. However, if the static program slice of an apparently correct variable contains faulty statements (that is, the apparently correct variable is in fact incorrect), these faulty statements may not be included in the resulting static dice. To safeguard against this problem, Lyle and Weiser (1987) proposed the following dicing theorem:

'A static program dice must contain the faulty statements (except for faults involving missing statements), when all the following conditions are satisfied:

1. The test suite is reliable.
2. There is one and only one fault (bug) in the program.
3. Suppose the computation of variable y depends on the computation of another variable x. If x has an incorrect value, then y has an incorrect value.'

A major drawback of static dicing is that it may contain non-executable statements which obviously do not contain the faults to be located. As an example of illustration, consider program 'Sample2' in Figure 2, which has a fault in function $f1$ of the *then* branch in $s3$. With the input ($x = -1$), output variable z is correct, but y is incorrect because y depends indirectly on the faulty function $f1$. Since the static slices with criteria (y,$s7$) and (z,$s7$) are $\langle s1,s2,s3,s4,s5\rangle$ and $\langle s1,s6\rangle$ respectively, the resultant static dice is $\langle s2,s3,s4,s5\rangle$. However, $s4$ is definitely not the cause of the fault, as only $s2$,$s3$,$s5$ are executed with this input.

Another drawback of static dicing is that it cannot properly handle dynamic data structures like arrays and pointer variables. In fact, this limitation applies to any static analysis technique.

3. DYNAMIC PROGRAM DICING

Since a static dice is constructed from the static slices of a correct variable and an incorrect variable, it involves two distinct variables. Thus, when all output variables are incorrect, the only alternative for the construction of a static dice is to use the static slice of a correct working variable. However, it may not be easy to verify the correctness of

```
      Program Sample2
      Begin
s1        read(x)
s2        If x < 0
s3            w = f1(x) /* bug ! */
          Else
s4            w = f2(x)
          Endif
s5        y = f3(w)
s6        z = f4(x)
s7        print(y,z)
      End
```

Figure 2. Program 'Sample2'

the working variable because normally the input–output relationship of working variables is not known. This motivates the study of the dynamic dice.

Although dynamic slices are input dependent, different inputs may give rise to the same dynamic slice. Hence, each dynamic slice is normally associated with a non–empty set of inputs rather than a single input. A dynamic slice is said to be correct if its slicing variable is correctly computed for all elements of this associated set of inputs; otherwise it is said to be incorrect. Intuitively speaking, a dynamic dice is the set difference between a set of incorrect dynamic slices and a set of correct dynamic slices.

We offer five dynamic dice construction strategies in this paper. Here is the simplest one that involves only one correct slice and one incorrect slice:

Strategy S1

A dynamic dice can be constructed by removing the statements of a correct dynamic slice from an incorrect dynamic slice.

Obviously, the variables for the correct and incorrect dynamic slices need not be distinct. However, dices constructed from the same variable are expected to be smaller than those constructed from different variables because the former are likely to have more common statements.

Consider program 'Sample3' in Figure 3, where variables y and z are output variables and statement $s6$ contains a faulty function $f2$. With input $I1$ = (n = 2, first x = −1, second x = 1), the dynamic slice with criterion ($I1$,$s9^{15}$,z) is $\langle s1,s2,s3,s4,s6,s7,s8\rangle$ and z is incorrectly computed. However, with input $I2$ = (n = 2, first x = −1, second x = −2), the dynamic slice with criterion ($I2$,$s9^{15}$,z) is $\langle s1,s2,s3,s4,s5,s7,s8\rangle$ and z is correctly computed. Hence, $\langle s6\rangle$ is the dynamic dice obtained from these two dynamic slices. This is an example of using dynamic slices of the same variable to define a dynamic dice.

For input $I2$, $\langle s1,s2,s3,s4,s5,s8\rangle$ is a correct slice with criterion ($I2$,$s9^{15}$,y) as y is correctly computed. If one takes the set difference between this slice and the incorrect slice of z above, we have $\langle s6,s7\rangle$. This illustrates the case of using dynamic slices of different variables to construct a dynamic dice.

```
      Program Sample3(Input n : integer)
      Begin
s1        i = 1
s2        While i ≤ n do
s3            read(x)
s4            If x < 0
s5                y = f1(x)
              Else
s6                y = f2(x) /* bug ! */
              Endif
s7            z = f3(y)
s8            i = i + 1
          Endwhile
s9        print(y,z)
      End
```

Figure 3. Program 'Sample3'

```
      Program Sample4
      Begin
s1        read(x,y,w)
s2        If x > 0 /* bug ! */
s3            y = f1(w)
          Endif
s4        z = f2(y)
s5        print(z)
      End
```

Figure 4. Program 'Sample4'

4. COMPARISON OF DYNAMIC AND STATIC PROGRAM DICING

4.1. General considerations

Obviously, a slice or dice is useful only if it contains the faulty statements. Unfortunately, a dynamic slice may miss some faulty statements. For example, in Figure 4, $s2$ should read 'if $x >= 0$'. For the input $x = 0$, $s2$ is evaluated to be false and hence $s3$ will not be executed. As a result, $s3$ is not the last definition of variable y used in $s4$ in this trajectory, although it should be. Hence, the dynamic slice with criterion $C = ((x = 0),s5^4,z)$, which is $\langle s1,s4,s5\rangle$, does not include the fault in $s2$. This problem is referred to as a 'potential influence' by Podurski and Clarke (1990).

When two dices contain the same faulty statements, obviously the smaller one is preferred because the smaller the size, the easier to locate the fault. Although a dynamic slice is included in its corresponding static slice, a dynamic dice is not necessarily smaller in size than the corresponding static dice. Program 'Sample5' in Figure 5 gives such an example. For the input $(x = -3)$, $\langle s1,s2,s6\rangle$ is both the incorrect static and dynamic slice of variable z; while the correct static and dynamic slices of y are $\langle s1,s2,s3,s4,s5\rangle$ and $\langle s1,s3,s5\rangle$ respectively. Hence the resulting static and dynamic dices are $\langle s6\rangle$ and $\langle s2,s6\rangle$ respectively. Obviously, the static dice is smaller in size than the dynamic dice.

If both the dynamic dice and static dice are available, then obviously the smaller one should be chosen for debugging. Since constructing dices may be very time consuming, we propose a probabilistic approach to determine their expected sizes from the size of

```
      Program Sample5
      Begin
s1        read(x)
s2        w = f1(x)
s3        If x > 0
s4            y = f2(w)
          Else
s5            y = f3(x)
          Endif
s6        z = f4(w) /* bug ! */
s7        print(y,z)
      End
```

Figure 5. Program 'Sample5'

the relevant slices. These expected sizes are used instead of the actual sizes to determine whether the dynamic or static dice should be used. Only the selected one needs to be constructed for debugging.

In this study, the three conditions of Lyle and Weiser's dicing theorem are assumed to be satisfied. In other words, both the dynamic dice and the static dice are assumed to contain all faulty statements. Thus, only the size is used as the criterion to compare the effectiveness of dynamic dicing and static dicing.

Unless otherwise specified, w and c are used to denote the incorrect and correct variables respectively in program $\mathcal{P}$. The static slices (dynamic slices) of w and c are denoted by SS_w and SS_c (DS^i_w and DS^j_c) respectively. We also denote the ratio of the size of the static (dynamic) slice of a variable x to that of the whole program $\mathcal{P}$ by SS_x (DS^i_x), where $0 < SS_x \leqslant 1$ ($0 < DS^i_x \leqslant 1$). No confusion should result as the context will clearly connote which definition is being referred to. Since a variable may have more than one dynamic slice, we use superscripts i and j to distinguish them. In this paper, we assume all statements have equal probability of being executed. Therefore, the expected sizes of the static and dynamic dices relative to the size of the whole program $\mathcal{P}$ are $SS_w(1 - SS_c)$ and $DS^i_w(1 - DS^j_c)$ respectively. It should be noted that the expected size is just a probabilistic estimate of its actual size.

For any variable x, α^i_x is used to denote the ratio of DS^i_x to SS_x. Obviously, α^i_x is in the range (0,1]. Since the maximum value of SS_x is 1, $\alpha^i_x \geqslant DS^i_x$.

When both α^j_c and α^i_w are equal to one, the static dice and dynamic dice are of equal size. Obviously, if α^j_c (α^i_w) is equal to one, dynamic dicing (static dicing) is better.

Proposition 1 states the necessary and sufficient condition for dynamic dicing to be better than static dicing, when both α^j_c and α^i_w are less than one.

Proposition 1

Suppose both α^j_c and α^i_w are not equal to one.

Dynamic dicing is better than static dicing iff

$$\frac{DS^j_c\,(1 - \alpha^{j^2}_c r)}{\alpha^j_c\,(1 - \alpha^j_c r)} < 1$$

where $DS^j_c/SS_c = \alpha^j_c$, $DS^i_w/SS_w = \alpha^i_w$ and $\alpha^i_w = r\alpha^j_c$.

Proof

$$\frac{DS^j_c\,(1 - \alpha^{j^2}_c r)}{\alpha^j_c\,(1 - \alpha^j_c r)} < 1$$

$$\Leftrightarrow \frac{DS^j_c\,(\alpha^{j^2}_c r - 1)}{\alpha^j_c\,(\alpha^j_c r - 1)} < 1$$

$$\Leftrightarrow DS^j_c \left(\alpha^j_c r - \frac{1}{\alpha^j_c}\right) > \alpha^j_c r - 1, \text{ since } \alpha^j_c r - 1 < 0$$

$$\Leftrightarrow 1 - \frac{DS^j_c}{\alpha^j_c} > \alpha^j_c r\,(1 - DS^j_c)$$

$$\Leftrightarrow \frac{DS_w^i}{\alpha_c^j r}\left(1 - \frac{DS_c^j}{\alpha_c^j}\right) > DS_w^i\,(1 - DS_c^j), \text{ since } \alpha_c^j r > 0 \text{ and } DS_w^i > 0$$

$$\Leftrightarrow SS_w\,(1 - SS_c) > DS_w^i\,(1 - DS_c^j)$$

$\Leftrightarrow$ dynamic dicing is better than static dicing. □

Proposition 2 states a sufficient condition for dynamic dicing to be better than static dicing.

Proposition 2

For any incorrect variable w and correct variable c, the expected size of a dynamic dice formed by DS_w^i and DS_c^j is smaller than that of the static dice formed by SS_w and SS_c if $SS_c < 1 - \alpha_w^i$.

Proof

Let $\alpha_w^i = r\alpha_c^j$.

$$0 < \alpha_w^i \leqslant 1 \text{ and } 0 < \alpha_c^j \leqslant 1$$

$$\Rightarrow 1 - \alpha_w^i \alpha_c^j < 1$$

$$\Rightarrow 1 - \alpha_c^{j^2} r < 1$$

$$\Rightarrow (1 - \alpha_w^i)\,\frac{1 - \alpha_c^{j^2} r}{1 - \alpha_w^i} < 1$$

$$\Rightarrow SS_c\,\frac{1 - \alpha_c^{j^2} r}{1 - \alpha_c^j r} < 1, \text{ since } SS_c < 1 - \alpha_w^i \text{ and } \alpha_w^i = \alpha_c^j r$$

$$\Rightarrow \frac{DS_c^j\,(1 - \alpha_c^{j^2} r)}{\alpha_c^j\,(1 - \alpha_c^j r)} < 1$$

$\Rightarrow$ dynamic dicing is better than static dicing after Proposition 1. □

During the testing of the program, a list of correct variables and their static slices would be generated. Suppose we decide to use dynamic dicing to debug w. Proposition 2 tells which correct variables favour dynamic dicing. However, there may be more than one correct variable satisfying the selection criterion imposed in Proposition 2. Based on the criterion of smaller expected size, the following propositions suggest which correct variable among those recommended by Proposition 2 is preferable.

Proposition 3

For any incorrect variable w, correct variable c and the dynamic slice being debugged DS_w^i, if $DS_c^j > DS_c^k$, then the expected size of the dynamic dice formed by DS_c^j and DS_w^i is smaller than that of the dynamic dice formed by DS_c^k and DS_w^i.

Proposition 4

For any correct variables c_1 and c_2 and incorrect variable w, if $MAX(DS^i_{c_1}) > \text{MAX}(DS^k_{c_2})$, then c_1 should be used to construct a dynamic dice with DS^i_w, where $\text{MAX}(DS^i_x)$ denotes the largest expected slice among all dynamic slices of x.

The proofs of Propositions 3 and 4 are omitted as they are trivial. Thus, for any program, if $\alpha^1_c = \alpha^2_c = \ldots = \alpha^m_c = \alpha^1_w = \alpha^2_w = \ldots = \alpha^n_w = \alpha$, there exist very simple conditions under which dynamic dicing is better than static dicing, and vice versa.

Proposition 5

Suppose $\alpha^1_c = \alpha^2_c = \ldots = \alpha^m_c = \alpha^1_w = \alpha^2_w = \ldots = \alpha^n_w = \alpha$, for the correct variable c and incorrect variable w. If the size of the correct static slice is less than half of the original program, dynamic dicing must be better than static dicing. However, the converse is not true.

Proof

By Proposition 1, dynamic dicing is better than static dicing if $\dfrac{DS^j_c\,(1 - \alpha^{j^2}_c r)}{\alpha^j_c\,(1 - \alpha^j_c r)} < 1$. If $\alpha^1_c = \alpha^2_c = \ldots = \alpha^m_c = \alpha^1_w = \alpha^2_w = \ldots = \alpha^n_w = \alpha$, then $r = 1$. Thus it follows from Proposition 1 that dynamic dicing is better than static dicing iff $DS^j_c \dfrac{(1 + \alpha)}{\alpha} < 1$, iff $SS_c(1 + \alpha) < 1$.

Since $f(\alpha) = \dfrac{1}{(1 + \alpha)}$ is a monotonic decreasing function and the domain of α is (0,1], the minimum value of $\dfrac{1}{(1 + \alpha)}$ is 0·5. Therefore, $SS_c < 0{\cdot}5$ implies that $SS_c < \dfrac{1}{(1 + \alpha)}$ and hence dynamic dicing is better than static dicing.

However, the converse is not true. The counter-example is the case that $SS_c = 0{\cdot}6$ and $\alpha = 0{\cdot}5$. In this case, DS^j_c becomes 0·3 and hence dynamic dicing is better than static dicing from Proposition 1 as $DS^j_c \dfrac{(1 + \alpha)}{\alpha} = 0{\cdot}9 < 1$. □

Proposition 6

Suppose $\alpha^1_c = \alpha^2_c = \ldots = \alpha^m_c = \alpha^1_w = \alpha^2_w = \ldots = \alpha^n_w = \alpha$, for the correct variable c and incorrect variable w. If the size of the correct dynamic slice is more than half of the original program, static dicing must be better than dynamic dicing. However, the converse is not true.

Proof

Similar to the proof of Proposition 5, static dicing is better than dynamic dicing iff $DS^j_c \dfrac{(1 + \alpha)}{\alpha} > 1$.

As $f(\alpha) = \frac{\alpha}{(1+\alpha)}$ is a monotonic increasing function and the domain of α is (0,1], 0·5 is the maximum value of $\frac{\alpha}{(1+\alpha)}$. Therefore, $DS_c^j > \frac{\alpha}{(1+\alpha)}$ as DS_c^j is greater than 0·5. Thus, static dicing must be better than dynamic dicing. The counter-example to show that the converse is not true is the case when $DS_c^j = 0{\cdot}45$ and $\alpha = 0{\cdot}5$. This implies that $DS_c^j \frac{(1+\alpha)}{\alpha} = 1{\cdot}35 > 1$ and therefore static dicing is better than dynamic dicing from Proposition 1. □

During the debugging phase of the development of a large application program, many static and dynamic slices need to be constructed. These slices, as well as their sizes relative to the program, are stored. Propositions 1 to 6 can use these data to provide estimates when the dynamic dices should be used in debugging instead of static dices, or vise versa. Hence, these estimates can serve as guidelines to help an automated debugger or its user to determine whether a dynamic dice or static dice should be used.

5. STRATEGIES FOR CONSTRUCTING DYNAMIC PROGRAM DICES

In this section, we present four additional strategies for constructing dynamic program dices. However, for simplicity, we will restrict our discussion to the case where only slices of the same variable are used for the construction of the dynamic dices. Extending this to using different variables is straightforward.

When the conditions of Lyle and Weiser's dicing theorem are all satisfied, a correct dynamic slice does not contain any faults. Hence, statements of such slices should not appear in the dice. Thus, Strategy S1 offered earlier can be improved as follows:

Strategy S2

Suppose we are now debugging variable x with dynamic slices $DS_x^1, DS_x^2, \ldots DS_x^n$, of which only the first k slices are correct. For each incorrect slice DS_x^i, $k < i \leqslant n$, a dynamic dice can be constructed as follows: $DS_x^i \backslash \cup_{j=1}^{k} DS_x^j$.

Obviously, if some of the conditions of Lyle and Weiser's dicing theorem are not satisfied, an apparently correct slice may contain faults. Hence, a dice may miss some faults. In this case, Strategy S2 may considerably magnify this undesirable effect and is therefore less effective than Strategy S1. Consider the case that variable x has 10 dynamic slices $DS_x^1, DS_x^2, \ldots, DS_x^{10}$. Suppose DS_x^{10} is the only incorrect slice that contains two faults, and that the other slices are apparently correct. Assume further that one of the faults appears in DS_x^1 and the other is in DS_x^2. If Strategy S1 is used, the dice produced by using only one of the apparently correct slices $DS_x^1, \ldots, DS_x^9$ will contain at least one fault. However, there will be no fault in the dice if Strategy S2 is applied. In this case, Strategy S1 is better than Strategy S2.

In practice, it is very rare to have all conditions of Lyle and Weiser's dicing theorem satisfied. Thus, we are interested in exploring strategies that can minimize the omission of faulty statements. One of our solutions is based on the following intuition:

The chance of having faults in the intersection of all correct slices is not greater than that for any individual correct slice.

We propose the following strategy which is aimed at minimizing the chances of missing faults:

Strategy S3

Suppose x has n dynamic slices $DS_x^1, DS_x^2, \ldots DS_x^n$, of which only the first k slices are apparently correct. For any of the incorrect slices DS_x^i where $k < i \leqslant n$, a dice can be constructed as: $DS_x^i \backslash \cap_{j=1}^{k} DS_x^j$.

Since the intersection of all correct slices should contain fewer faults than any single correct slice does, Strategy S3 has less chance of missing faults than Stragegy S1. As a reminder, every dice constructed by Strategy S1 is included in some dices constructed by Strategy S3.

We offer another intuition that is complementary to the one mentioned above:

The density of faults should be higher in the intersection of all incorrect slices than the density in any individual incorrect slice.

We propose the following two strategies based upon that intuition:

Strategy S4

For any correct slice DS_x^j, where $1 \leqslant j \leqslant k$, a dice can be constructed as: $\cap_{i=k+1}^{n} DS_x^i \backslash DS_x^j$.

Strategy S5

By making use of all dynamic slices of x, a dice can be constructed as: $\cap_{i=k+1}^{n} DS_x^i \backslash \cap_{j=1}^{k} DS_x^j$.

Since the intersection of all incorrect slices must be included in every individual incorrect slice, the dice constructed by Strategy S4 or S5 cannot be greater than that constructed by Strategy S1 or S3 respectively.

Strategies S4 and S5 do not guarantee that their dices would not miss faults, unless there is only one fault in the program. In fact, Strategy S4 is not designed to minimize the chance of missing faults. It should be noted that the omission of faults usually does more harm than the reduction in dice size does good. There are two ways to overcome this problem. First, instead of using slices of the same incorrect variable along different paths, slices of different incorrect variables but along the same path should be used. The rationale is simply that it is more likely for the incorrect slices derived from the same path to contain the same fault. Hence, the intersection of these incorrect slices should be less likely to omit the fault. Second, a trial and error approach can be applied to select a subset of incorrect slices for intersection until the generated dice helps the debugging.

For Strategies S2 to S5, when dynamic slices of various variables are used, they should be chosen along the same execution path (that is, dynamic slices of various variables for

the same input) rather than from different execution paths. The rationale is that slices from the same execution path should have more common statements. Hence, Strategies S3 and S5 should yield a smaller dice because there are more common statements in the correct slices. On the other hand, this also favours Strategies S4 and S5 because there is a higher chance that slices of incorrect variables along the same path have the same source of error. Therefore, the intersection of these incorrect slices is more likely to contain the same faults and, hence, so is the constructed dice. Furthermore, when only one execution path is involved, there should be less computations for constructing the slices.

The major computations in the application of Strategies S2, S3, S4 and S5, are the extensive constructions of correct and incorrect dynamic slices. Since the dynamic slices can be obtained as a by-product of a dynamic testing system, it is natural to integrate the techniques of dicing and slicing with dynamic testing.

6. DISCUSSION AND CONCLUSION

In this paper, we have proposed dynamic dicing as the dynamic counterpart of static dicing introduced by Lyle and Weiser (1987). We observe that dynamic dicing has the following advantages over static dicing:

1. when there is only one static slice (that is, only one output variable) or all static slices are incorrect, static dicing cannot be applied, but dynamic dicing can be;
2. a static dice may contain non-executable statements, but in a dynamic dice, all statements are executable; and
3. dynamic data structures, like arrays and pointers, can be handled by dynamic dicing but not by static dicing.

A probabilistic approach has been used to investigate the relationship between the sizes of the static and dynamic dices. The results of our study provide guidelines to determine when dynamic or static dices should be used, even prior to their construction.

Based on the size of the dice and the chance of omitting incorrect statements in the dice, we have developed several dicing strategies. However, none of them is best under all circumstances.

Since different execution paths may yield the same dynamic slice, there will be redundancy in constructing dynamic slices. A possible solution is to generate the static slice of the incorrect variable first, then decompose it into all possible dynamic slices. However, such a decomposition algorithm needs further investigation.

In the research reported here, we took an approach that was either purely dynamic or purely static. It should be noted that a hybrid approach would yield a smaller dice if it is defined as the set difference of an incorrect dynamic slice and a correct static slice—that is, $DS^i_w \backslash SS_c$. However, $DS^i_w \backslash SS_c = (SS_w \backslash SS_c) \cap (DS^i_w \backslash DS^j_c)$ because $DS^i_w \subseteq SS_w$ and $DS^j_c \subseteq SS_c$. In other words, $DS^i_w \backslash SS_c$ is equal to the intersection of a dynamic dice and a static dice. Since two different variables are involved in this hybrid approach, we can only establish at this time that it is better than the static dice $SS_w \backslash SS_c$ or the dynamic dice $DS^i_w \backslash DS^j_c$ where w and c are not the same variable. However, $DS^i_w \backslash SS_c$ may not be smaller than the dice $DS^i_w \backslash DS^j_w$ where DS^j_w is a correct dynamic slice.

Acknowledgements

This paper is an improvement on and extension of the work we reported in Chen and Cheung (1993). We would like to express our gratitude to F. T. Chan, T. H. Tse, L. J. White and Y. T. Yu, for their helpful and invaluable discussions. We also would like to thank M. F. Lau for his invaluable comments leading to Proposition 5, and are greatly indebted to G. Eddy and J. Shepherd for their invaluable discussions and advice on improving our presentation in this paper.

References

Agrawal, H. and Horgan, J. R. (1990) 'Dynamic program slicing', *SIGPLAN Notices*, **25**(6), 246–256.

Chen, T. Y. and Cheung, Y. Y. (1993) 'Dynamic program dicing', in *Proceedings of the Conference on Software Maintenance—1993*, IEEE Computer Society Press, Los Alamitos, CA, pp. 378–385.

Gallagher, K. B. and Lyle, J. R. (1991) 'Using program slicing in software maintenance', *IEEE Transactions on Software Engineering*, **IEEE–SE–17**(8), 751–761.

Gopal, R. (1991) 'Dynamic program slicing based on dependence relations', in *Proceedings of the Conference of Software Maintenance—1991*, IEEE Computer Society Press, Los Alamitos, CA, pp. 191–200.

Howden, W. E. (1976) 'Reliability of the path analysis testing strategy', *IEEE Transactions on Software Engineering*, **SE–2**(3), 208–215.

Kamkar, M.; Fritzson, P. and Shahmehri, N. (1993) 'Interprocedural dynamic slicing applied to interprocedural data flow testing', in *Proceedings of the Conference on Software Maintenance—1993*, IEEE Computer Society Press, Los Alamitos, CA, pp. 386–395.

Korel, B. and Laski, J. (1988) 'Dynamic program slicing', *Information Processing Letters*, **29**(3), 155–163.

Korel, B. and Laski, J. (1990) 'Dynamic slicing of computer programs', *Journal of Systems and Software*, **13**(3), 187–195.

Lanubile, F. and Visaggio, G. (1993) 'Function recovery based on program slicing', in *Proceedings of the Conference on Software Maintenance—1993*, IEEE Computer Society Press, Los Alamitos, CA, pp. 396–404.

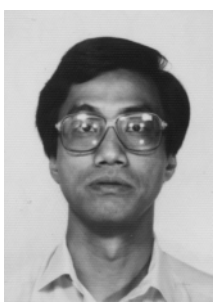
Lyle, J. R. and Weiser, M. (1987) 'Automatic program bug location by program slicing', in *Proceedings of 2nd International Conference on Computers and Applications*, IEEE Computer Society Press, Los Alamitos, CA, pp. 877–883.

Ott, L. M. and Thuss, J. J. (1989) 'The relationship between slices and module cohesion', in *Proceedings of the 11th International Conference on Software Engineering*, ACM Press, New York, NY, pp. 198–204.

Podurski, A. and Clarke, L. A. (1990) 'A formal model of program dependences and its implications for software testing, debugging and maintenance', *IEEE Transactions on Software Engineering*, **SE–16**(9), 965–978.

Weiser, M. (1984) 'Program slicing', *IEEE Transactions on Software Engineering*, **SE–10**(4), 352–357.

Authors' biographies:



T. Y. Chen received the B.Sc. and the M.Phil. from the University of Hong Kong; the M.Sc. and the D.I.C. in computer science from the Imperial College of Science and Technology; and the Ph.D. in computer science from the University of Melbourne. He is currently a senior lecturer in the Department of Computer Science, University of Melbourne. His main research interests include fixpoint theory, program testing, software maintenance and software engineering.

Y. Y. Cheung received the B.Sc. in computer science from the Chinese University of Hong Kong, and is currently a postgraduate in the Department of Computer Science, the University of Melbourne, where she is also presently a member of the Software Testing Group. Her main research interest is software maintenance, with emphasis on software testing and debugging.